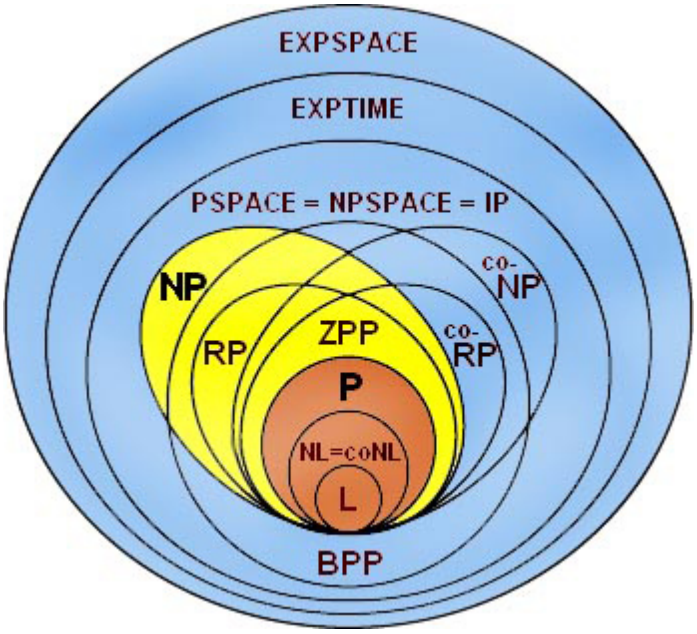
彎曲評論
科技 · 人物 · 潮流

浅谈高端CPU Cache Page-Coloring

作者：陈怀临
huailin@gmail.com



前言：

本文通过读者们比较熟悉的妈咪和包厢场景，阐述了高端CPU和大Cache结构中的一个比较略微高深的工程话题 - - Cache Coloring。高端通信系统设计和实践中，对Cache和性能的把握是至关重要的。希望这篇文章对大宋的读者，特别是熟悉妈咪的读者有所帮助。世界上的再高深的技术和理论都可以在现实生活中找到映射。这就是所谓的：得其意；忘其形。方为做学问之最高境界了。

OS和体系结构的关系是：OS是形而上；体系结构是形而下。一个高手对这两面都必须有相当得修养，缺一不可。希望这篇文章能起到抛美女引客人的目的，使得读者们能够在业余时间去体验生活，阅读相关资料。提高自己的各方面的能力。。。。。。

浅谈高端CPU的Cache Coloring（1）

现代CPU是越来越狠。x86+PCI-E 3似乎要横扫一切的样子。Tilera等高端CPU也咄咄逼人。
RMI 832也似乎在朝着大型CPU的方向靠拢。换言之，经典Server和Desktop CPU与网络CPU在融合，convergence。 你中有我；我中有你。
不久的将来，Software-Based 各种设备，估计顶到Edge的层面，似乎都不奇怪。
但，要把大型CPU用的好，绝非写写胶片，就能搞定。
胶片一分钟；台下N年功。
其中，对大型CPU的大Cache的有效利用是其中重中之重。
这就好比，你忽悠了一个美女到家里，但是美女发现你不能很好的满足她的各种需求【此处省略249字。。。】
观察业界的各种大CPU，最凸显的就是L2，L3 Cache的巨大。而且L3都已经在Die里。
对桌面和服务器系统来说，这些Cache的管理基本上与业务无关。OS基本上全Cover了。
但对于通信系统，必须aware。
为啥涅？
通信系统的Data Path（含数据结构）是非常要求realtime的。是线速要求的关键之处。
友善的Cache命中率可以使得一个系统的性能让人不相信，让人觉得是一个863的成果，或者核高基项目的汇报演出：-）。
那么如何分配数据结构？ 如何Cache Friendly？
有许多方法，但其中一个重要的手筋就是–Page Coloring
Page Coloring方面比较绕人。要搞懂，基本上需要对OS和CPU都明白。要能真正的敢用，估计江湖上也就陈首席一个人了。。。。。。要用好，目前没看见 过。 Panabit系统估计在Cache方面用的比较好。但是否用了Page Coloring目前看不出来。

什么是Page Coloring？

首先不要上来就去看学术文章。

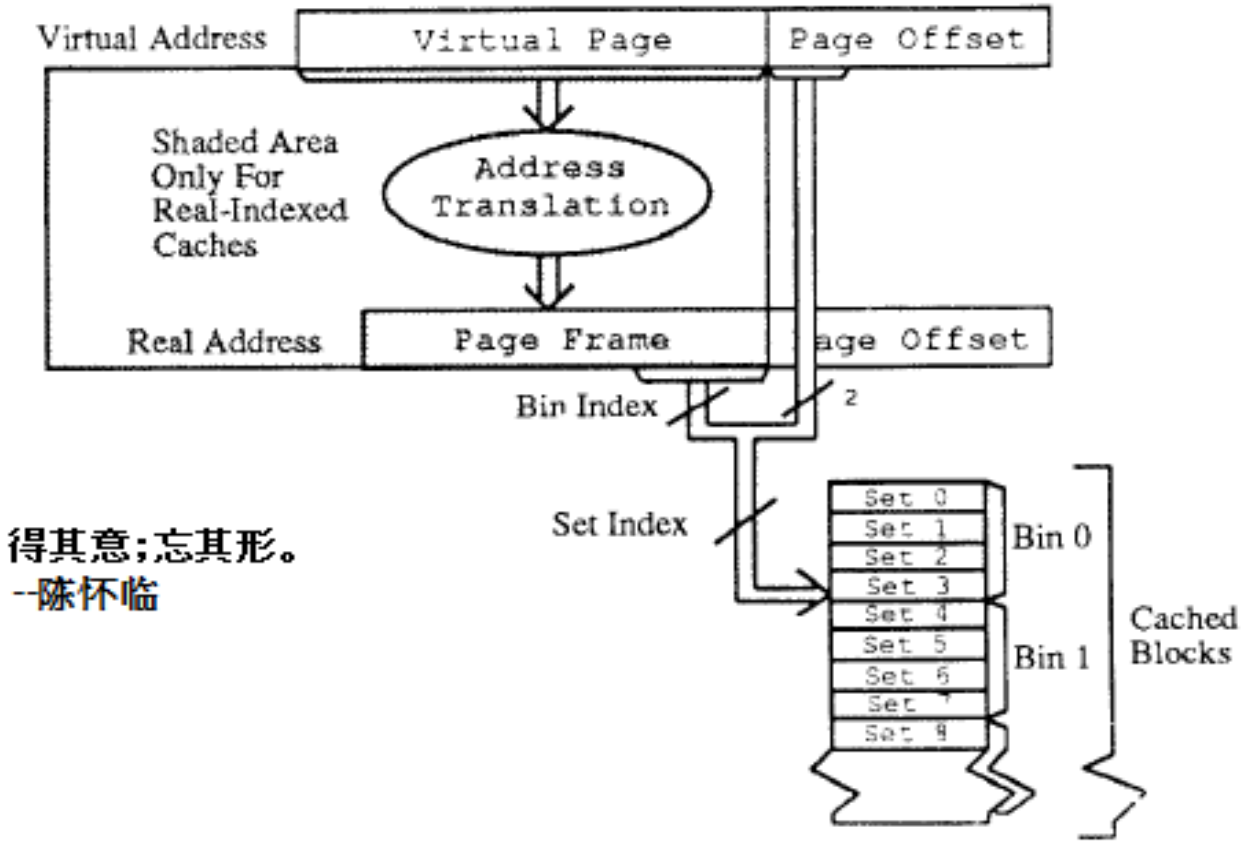
最好的学术一定是来源于生活。例如，陈首席的多核与公厕的悖论原理。

× Page Coloring其实就是要让OS在内存管理方面Cache Aware。Cache感知。

× OS VM对物理页面的分配机制与CPU Cache的管理机制不是和谐的。

×Page Coloring的前提是这个CPU必须是Physically Indexed。

下面通过一张精选的图示，来看看什么是Page Coloring。



得其意;忘其形。
--陈怀临

在大多数OS的VM管理中，当分配一个页面Page[Note: 一定要牢牢记住，Page的概念是OS的概念，而非CPU的概念。这个问题出现混淆，请买豆腐撞死】，通常是4K一个页面。这里是否是4K不重要，就是一个happen to 4K而已。

如果是4K，如何分配的？

一定是0-11 bits是Offset（2^12=4K）。31-12bits是Page Number。这就是上图的所谓Page Frame。

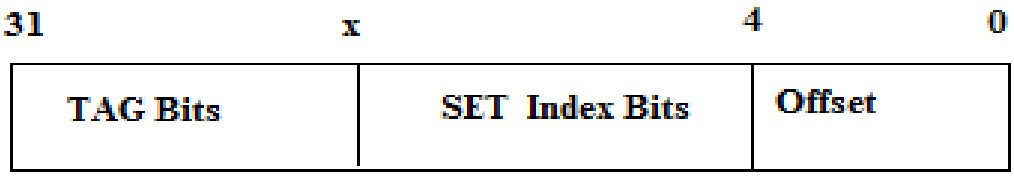
那么请问，这些一个个Physical的Page是如何落入Cache的手里的？

这个问题的等价问题是（用洋文说是，reduced to）：

Cache是如何Allocate，Hit的？

现代CPU基本上都是Set-Associative的管理方式。

现代CPU Cache Index方案



这种Cache方案意味着：

×每个Cache Line的大小是2^5= 32Bytes
×系统中有2^(x-4)个SET。
×系统Cache大小是

一个Y Way的Cache，
Cache Size = Y * 2^(x-4+5)= 2^(x+1) . Y

上述运算是在知道一个Cache是多少Way，有多少SET和Cache Line大小，算Cache Size。

反之，知道任何两个参数，都能换算出另外一个参数。例如知道一个Cache 大小是X; Way是Y; Cache Line大小是Z。那么SET的值是：

SET = x/(y*z)

我们来简单的看看一个大Cache的Set-Associative的管理方式是如何与经典OS的Page管理方式脱节的。

这种脱节是导致了Page Coloring算法出现的唯一动机。

算法的胶片是要解决问题的。否则就是YY（意淫）。

例如一个1M的Cache Line大小为32Byte的8Way Set Cache。

8Way的意思是：一个SET（集合）有8个Cache Line。

那么这个CPU能有多少个SET？

1M/(32*8) = 2^20/2^8 = 2^12 = 4K = 4096

这个CPU有4096个SET。每个SET是8个Cache Line。

一个问题涌现了： 如何使得OS层面上物理页面（Page）的分配能够比较均匀的落在CPU层面上的SET中。从而可以避免自相残杀和互相挤兑？

上述命题可以reduce成为这样一个命题：

一个基于4K为页面大小的OS层面的Page分配机制与一个基于32 byte Cache Line大型的SET -

Assoc分配机制的Cache管理基本上没有任何逻辑关系？
但这种关系的弱映射是至关重要的。
我们如何来解决？如何把一个OS与底层的CPU在语义上来紧耦合？
最关键的地方出现了。
就是上述我辛勤画出的这个x的值。x的位置。

浅谈高端CPU的Cache Coloring（2）

为了理解Page Coloring，需要把握或者记住下面几点：
× OS的内存管理的基本分配粒度是Page，例如经典的4K大小；
×CPU的Cache管理的基本上分配粒度是基于Way-SET的Cache Line。例如，32bytes。
×大Cache，例如，L2，L3，很大，很大。
在上述 3 个前提下，如果理解Page　Coloring？
思考了好几天，如何用大白话来说，而非玩学术。玩胶片。
我的理解大概是这样的：
理解一：大宋姐妹一盘棋
如果北京八大胡同的姐妹们和东莞的姐妹们都是我大宋服务业的一盘棋，我们就要一盘棋来考虑，要拉通【注：某司的专门术语。好像意味着互通有无，或者有一个 良好的通信Channel】。 因此，不要都往北京的天上人间或者地下人间跑，我们可以定一个政策，南方人就去东莞折腾；北方人就在北京。再整一个成都，西北人都去成都耍。Page Coloring其实就是这个意思。南方人，北方人，西北人就是所谓的识别着色（Coloring）。这样的好处是啥涅？不会导致大家都往成都跑【我的 YY呀。没去过。成都的应该好点吧。北京的姐妹们的一口东北口音让人基本上无言以对，情何以堪呀】。
亮点： 相对均匀的分配落脚点的好处是都整体经济拉通有好处。

理解二：不要输在起跑线上
现在凡是都要计划。据说小朋友还没有出身，妈妈们就要开始琢磨上什么亲子班和各种攻略。为了就是让孩子们有个好的落脚点。可怜天下父母心。出身论在我大宋 是根深蒂固的。红N代估计从来就看不起贫M代。富K代估计从来都忘了自己的贫K-j的祖先。如果不输在起跑线上？ 就是planning，把自己的孩子往中关村幼儿园整；往深圳的实验小学整。如何整？ Coloring！ 户口，买房子的地点等等。。。。
亮点：人为调整住房和户口【含假离婚（据说最后都变成真离婚）】信息，映射到最好，

或者次好的校区。

Coloring的目的就是把一个东西的落脚点，有目的地，设置好，从而获得最大利益。这个设置就是通过在可控的范围内（OS），把物理Page的Page Frame信息调配好，从而最大程度上的利用大Cache。

现在我们来看看CPU的Page和Cache是如何协同工作的。

现在考虑一个L2 Cache，参数如下：

–大小： 2M

–SET/Way： 4 Way SET

–Line: 32 Bytes

这个2M的L2 Cache有多少SET（集合或者组）？

2M/（32 × 4） = (2 × 2 ^20)/ (2^5 * 2^2)=2^14=16K=16*1024

=10240+6144=16384

这个2M的cache是分成了16384个SET。每个SET里含有和管理4个Cache Line。这个管理算法可以是P-LRU，当然，也可以是王大师的WLRU算法。这个SET里面的替换算法在此不讨论。基本上类似于贵族幼儿园内部的潜规 则。例如，送礼多的，老师就照顾多点。或者不送礼的，下个学期突然就让你的孩子out了，被另外一个小replace你们家小孩的名额了。

我们现在来看一个OS层面的Page，4K大小。是如何落在这16384个SET里的。

不失一般性，我们假设这个Physical Page是0×0. 最低端的4K内存。

其地址范围是：0 - （ 4K - 1 ）。

我们来考察其在这个2M Cache中的分布情况。

问题一： 一个Page有多少Cache Line？

4K / 32 = 2 ^ 7 = 128。

也就是说每个OS层面分出来的物理页面会占据128个CPU层面的Cache Line。

现在假设我们对这个4K的区域做一个memset（0x0，0， 4 * 1024）.

显然，在2M Cache的CPU下，这个4K的内存一定都会被带到Cache中来。

但是如何分布的？

是散落在这16384个SET中的某一个连续的128个SET中的。这里的某一个，其实就是第一个128个SET。

上述的这段话要绝对的理解清楚。否则下面的图都无法看，和对Page Coloring无法理解。

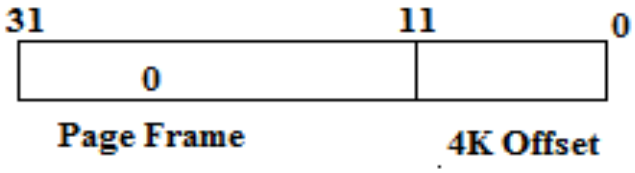
一个4K大小的物理内存的东西被按照每32字节大小放在了128个不同的SET中

reduced to：

4千个贪官，来自不同的机关单位。每个单位是32个人，排队去天上人间（北方干部），

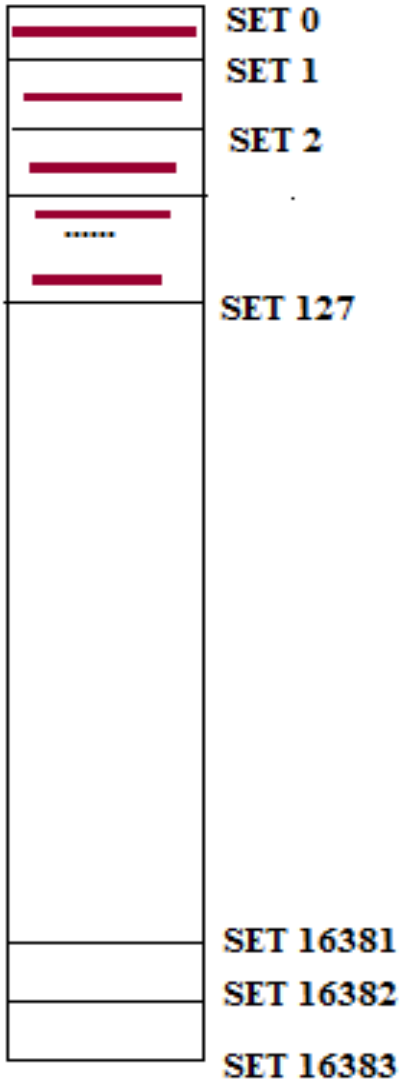
或者去东莞（南方干部），或者去成都（西部干部）。
妈咪是这样处理的：

× 来自同一个机关的干部们（32人一组）都在一个包厢（SET）里。
【妈咪注：每一个包厢其实可以容纳4组（32×4）贪官。但另外3组席位那是为另外贪官队伍准备的。目前，就只放一组进来】
×不同单位的干部们都在相邻的前后左右包厢里。
×4千个贪官整了128个包厢

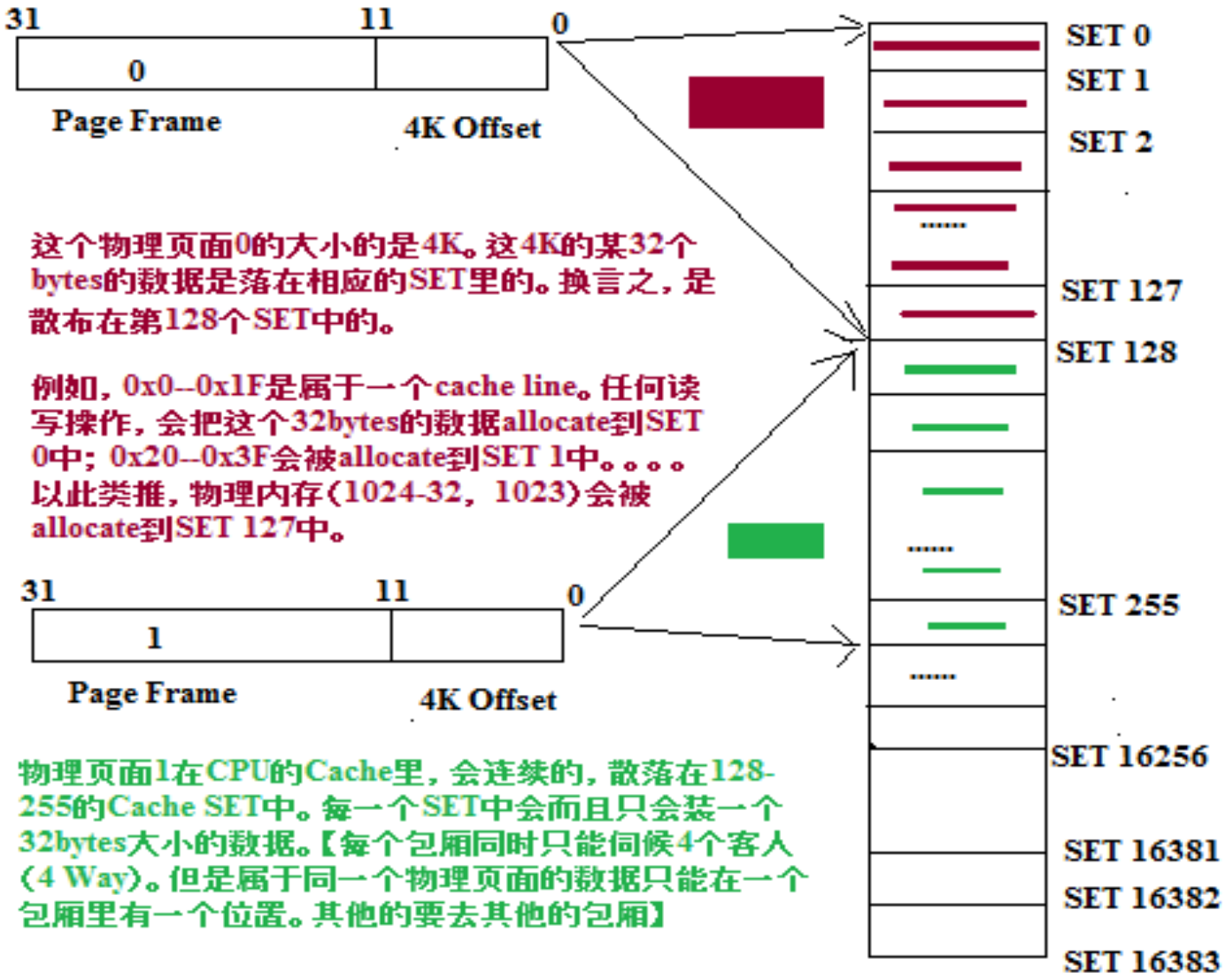


这个物理页面0的大小的是4K。这4K的某32个bytes的数据是落在相应的SET里的。换言之，是散布在第一个128个SET中的。

例如，0x0--0x1F是属于一个cache line。任何读写操作，会把这个32bytes的数据allocate到SET 0中；0x20--0x3F会被allocate到SET 1中。。。。以此类推，物理内存(1024-32，1023)会被allocate到SET 127中。



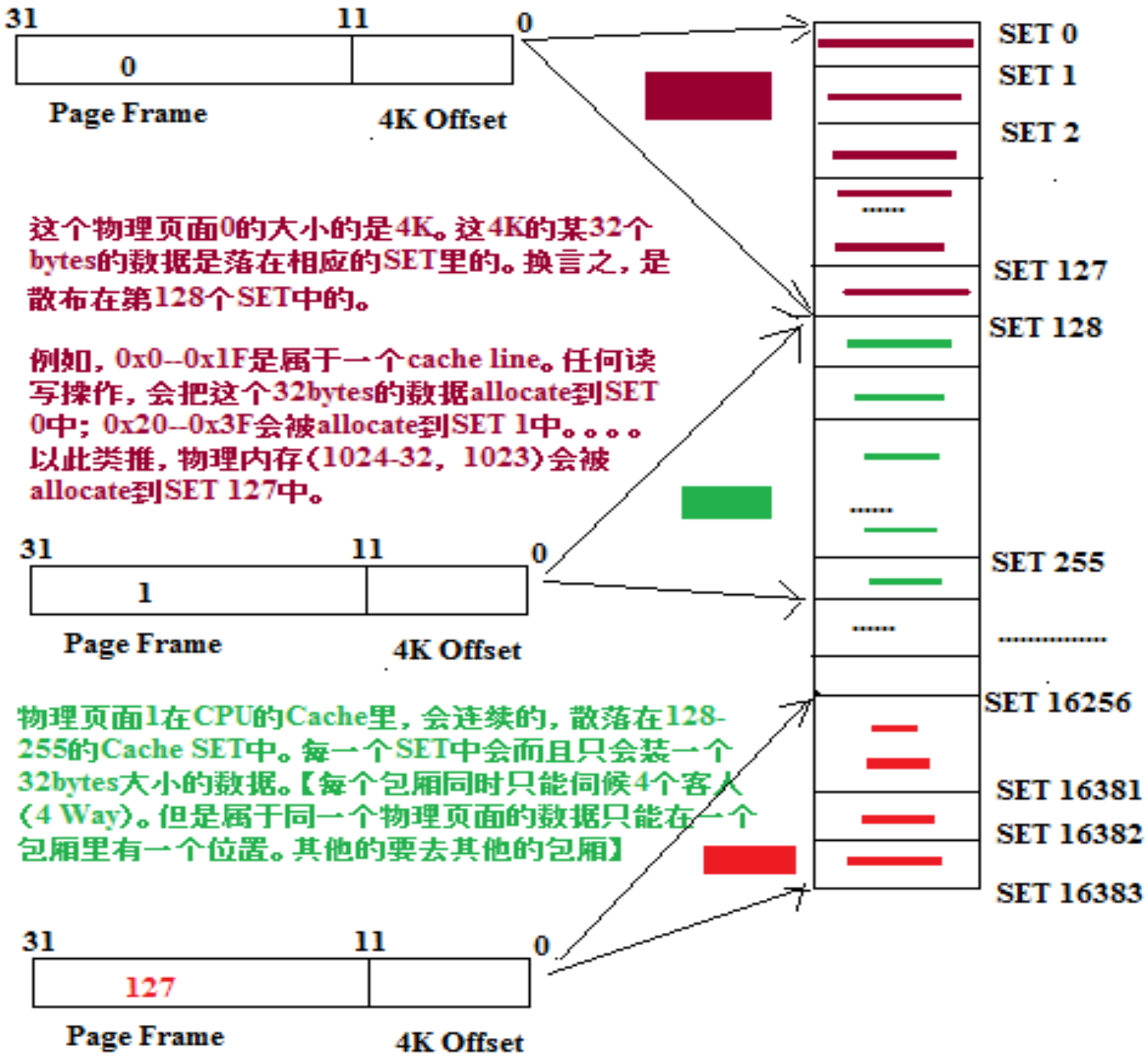
那么物理页面 1 是如何在Cache中分布的？



以此类推，第3，4，5个物理页面也都会被分配和跟在后面的Cache SET 中，而不会与前面的物理页面占据的Cache位置相冲突。

Until 第（16386/128=2^14/2^7=2^7=128）个物理页面被分配之后。

换言之，如果我们连续对128个物理页面做memset清0，就会把这个2M的L2 Cache的每个SET中放上一个Cache Line。而且没有任何冲突。



问题就是出在128个物理页面之后的事情。 事情要绕回去，从头来了。

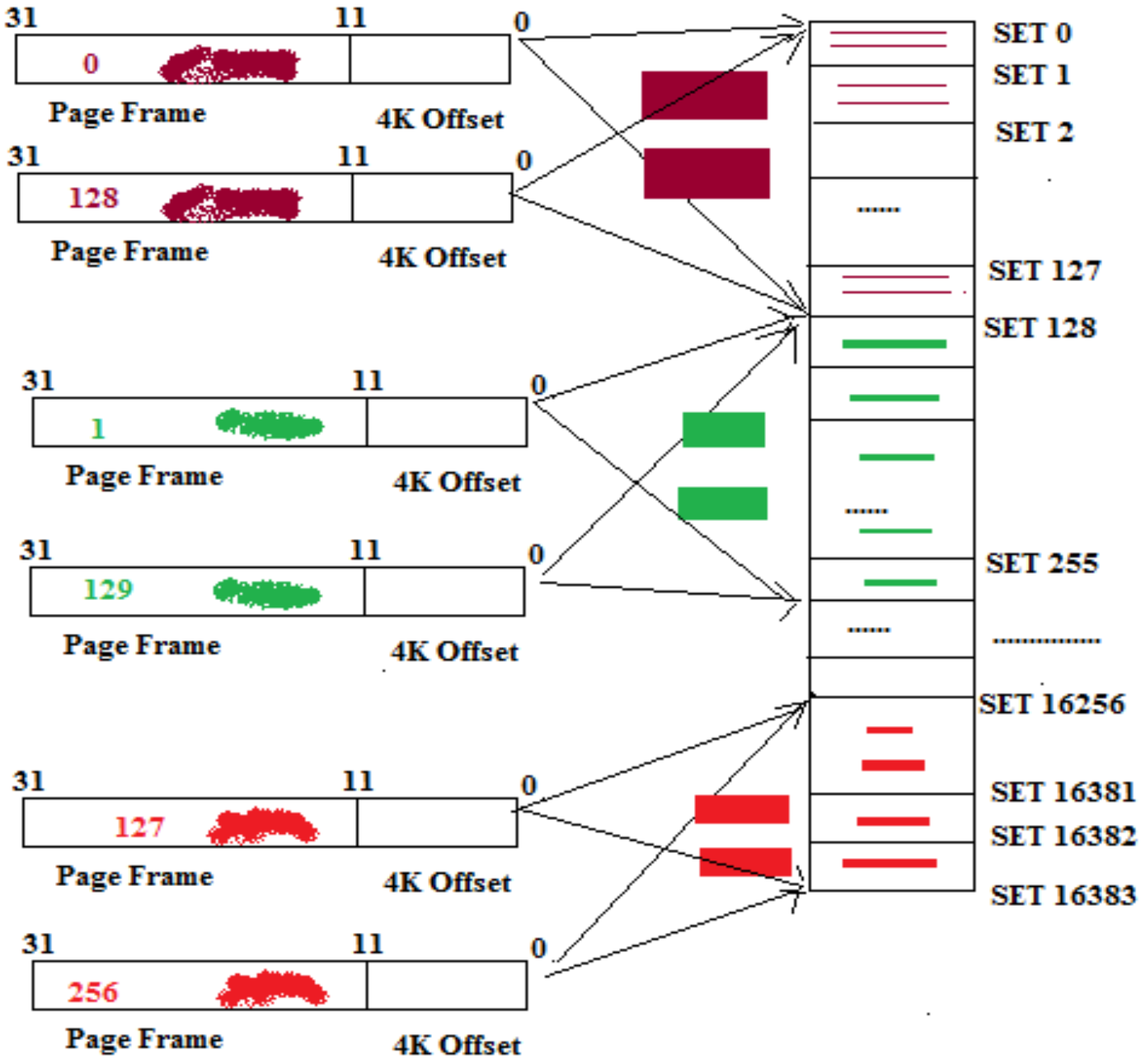
为什么？

浅谈高端CPU的Cache Coloring（3）

上节中我们讲到，如果对内存的0-127（总共128）个物理页面做内存清零（memset）的话，其在cache中的分布是正好每一个SET（包厢）里，放入了一个cache line（包厢里的长凳子。每个凳子上32个人。每个包厢是4个凳子：4 Way）。
现在来考虑，接着对第128th 物理Page清零。我们来考察其在Cache中会落在哪里。
我们先谈结论。
结论： 第128th物理页面会与第0个页面占据同样的Cache Sets。
推论：
*** 第ith物理页面会与第（128+i）th个物理页面在Cache中占据同样的Cache Sets。**
*** 第ith物理页面会与第（128*j+i)th个物理页面中Cache中占据同样的Cache Sets。（i,j = 0,1,2….)**

属性讨论：
× 占据同样的Cache Sets，会导致潜在的Cache冲突，替换。例如，当一个N-Way的Set，如果所有的Way（长凳子）都被坐满的时候，必须按照某种算法，把倒霉蛋踢出局。
× 我们把i, 128×j+i(i，j=0,1,2,3…)等属于同一个Cache Sets映射的物理页面，称为具备同样颜色（Color）的物理页面。
×请注意，每一个4K的物理页面上映射到127个连续的SET中。是Cache SetS。学术上，这个相应的容纳这个4KPage的Cache SET的集合，叫做 Cache Bin。

下面为相应的图示：

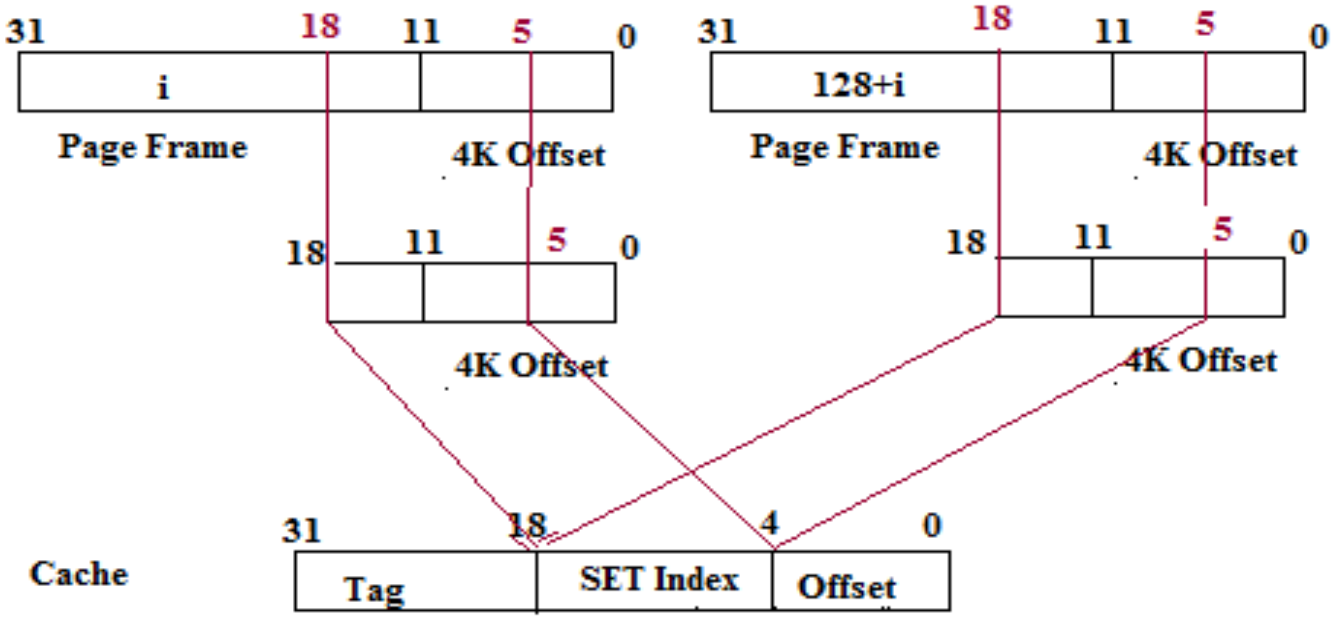


陈怀临注：系统cache set中的data分布图，当第二个128个物理页面被读进Cache。这时，由于L2的每一个SET是4 way【每个包厢有四个长凳子；每个凳子可以坐32个人；每次进出都是32人（cacheline）】，所以现在每个包厢还不需要赶人，还有两个凳子。。。。。。

下面来分析一下为什么第ith个物理页面与第（128*j+i）个物理页面会是映射到同样的cache set中。

我们讨论的这个Cache是一个L2 Cache，参数如下：

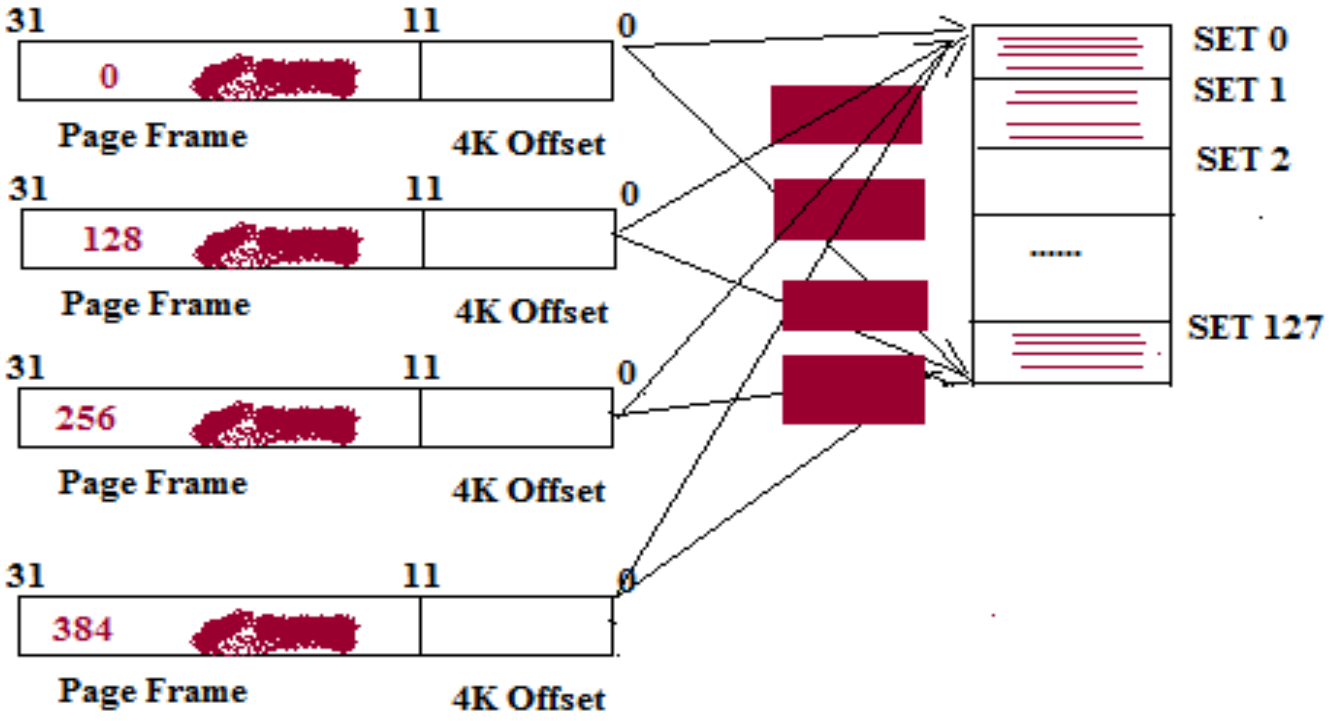
–大小： 2M –SET/Way： 16384/4 Way SET –Line: 32 Bytes
16K的SET。意味着2^14。
换言之，需要物理地址贡献出14个bit，来做为定位其落在哪个Cache Set.



从上图我们知道，bit 5-18是用来选择属于哪个SET的。
× 由于一个物理Page的5-11是可变的【4K的物理Page】。因此这就是为什么上节所解释的一个物理Page一定是占据了连续的128个SET中的某一个cache line： 11-5+1 = 7. 2^7= 128.
* 对于一个已知的i，其bit 12-31【共24bit】是已经固定的，fixed的了。是不变的。

现在来考察 i 与128*j+i的关系。
很显然，物理页面i与物理页面i+128*j的低7位数完全一致的。换言之，bit 12-18是完全一致的。
例如，
128-0 = 128 = 0b1 0000000;
128*1 -0 = 0b10 0000000;
128*2 – 0 = 0b11 0000000;
……
以此类推，

128×j – 0= j(128-0) = j * (0b 1 0000000);
= 0b x 0000000;
其中 x的十进制值为j。
由此可见对于任何一个物理页面i，对应的128+i页面，128×2+i，128×j+i页面的bit 12-18是相同的。
这意味着什么？
请注意，Cache的SET的选择就是通过5-18bit来决定的。
如果两个物理页面的12-18bit是一样的，
这意味着： 这些页面的4K内容都会落在相同的128个SET中（5-11位决定的）。这些页面属于一个相同的颜色（Color）。
下图所示是当4个具有同样颜色的物理页面都被读到Cache中后，Cache的SET和Way的情况–“属于你的”包厢的4条凳子都全部坐满。
这里面，“属于你的”这个词非常重要。即使妈咪有别的包厢，但如果不属于你，你是不行的。有钱都不灵。“属于你的”其实就是你的颜色；你的颜色决定了你的出路和你的地位；这有点类似与当年，你的阶级成分决定了你许多的将来。。。。。。由于出身论被枪毙的遇罗克同学是一个英雄。。。。。。

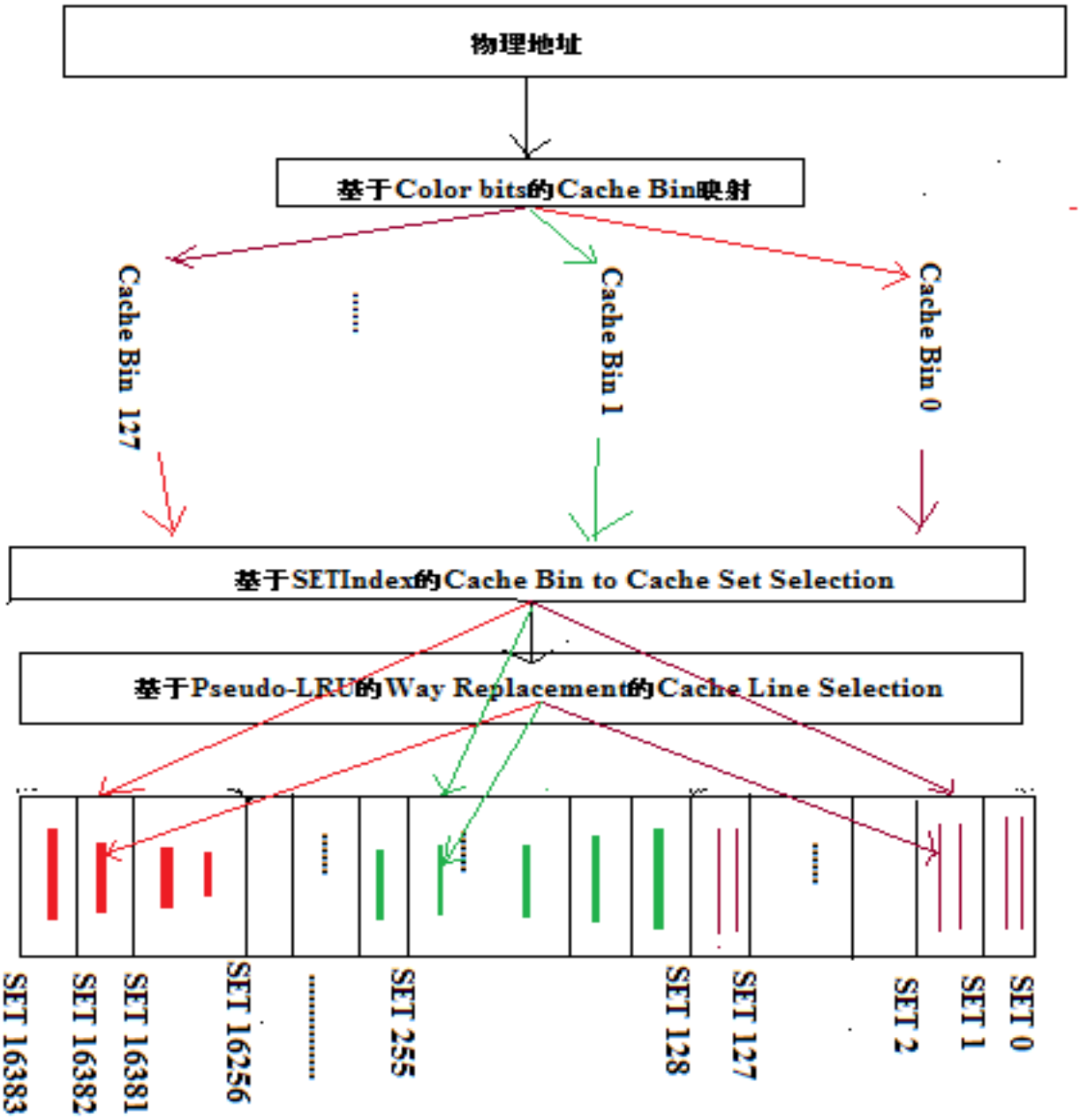


陈怀临注：4个Physical Page(4×4=16K)，就会把第128个SET的所以Way全部布满(128×4×32)=2^14=16K。
之后，任何新的Cache Line要进来，就必须寻找牺牲品了。。。

浅谈高端CPU的Cache Coloring（4）

CPU Cache的Page Coloring相对而言算OS和【或】系统方面比较高深一点的话题。涉及的技术其实不多，但涉及的概念比较广和杂乱。
为什么我竭尽全力的试图用大白话来把这个专题谈清楚呢？
市场原因如下：
1. 我个人Believe，基于x86或者高端NPU，具备大cache的通信设备会对中低端市场造成倾销似的冲击。 Software Based系统会从接入一直顶到Edge。
2. 我国大量企业缺乏FPGA和ASIC的能力。利用通用CPU，迅速的切入市场，同时可以利用Linux丰富的3rd party的应用。可以迅速的占领市场份额。
3. 对Software Based系统，能把系统做好做精，cache的把握是very important, if not most important.
技术原因如下：
1. 在OS和体系结构的各种经典书籍中，基本上对Cache Page Coloring涉及不多。或者没有提及。同学们在这方面的知识结构需要略微弥补。
2. 对Page Coloring的理解，类似与造大飞机一样：造大飞机重要；但更重要的是，可以拉通一系列产业。通过Page Coloring，可以使得同学们弥补和温习OS和体系结构的许多概念和知识，从而使得大家能够成长。
3. 我自己在看了许多文献之后，有时也是很迷糊。也单独请教过Xiaodong Zhang。感觉Page Coloring的文献都写的比较学术化，处于风花雪月的小圈子里。而且感觉在学校里的教授和博士生，基本上缺乏实战经验，写的东西有点虚。但随着高端 CPU的不断应用，工程技术人员，特别是，公司里的系统骨干人员，有必要了解和even掌握。
从这节开始，我会试图对page coloring开始做定性和定量的分析（Qualitative and Quantitative Analysis of Cache Page Coloring)。
要掌握Page Coloring，首先大家要从概念补起。
下面是我个人阅读和参阅的一些OS和体系结构的书籍：
×Unix Systems for Modern Architectures -Curt Schimmel
×Unix Internals -Uresh Vahalia
×Computer Architecture-A Quantitative Approach -Hennessy & Patterson
×The Art of Multiprocessor Programming -Maurice Herlihy et. al.,
×The Design and Impl of the FreeBSD OS -Marshall Kirk et. al.,
×Computer Systems-A Programmer's Perspective – Randal Bryant et. al.,
×Understanding the Linux Kernel – Daniel Bovet, et. al.,
上述书籍基本上涵盖了OS和体系结构的概念，知识和相关技能。但基本上对大Cache的Page Coloring是篇幅很少, if not ZERO。

这里面的原因有三：
1. Page Coloring只能在大Cache的情况下才能出现，例如L2，L3 Cache。对L1 Cache基本上没有任何意义。
2. Page Coloring是一个OS与体系结构相碰撞时，而产生的问题。是一个介于工程和学术之间的问题。
3. 作者要么来自OS方面，要么是来自体系结构方面。很难有时间和精力在这个交叉的专题方面专门讨论。
为什么Page Coloring容易迷糊人？
结果如下：基本上所有的教科书没有introduce一个重要的概念给学生。从而使得，学生的知识结构出现了一个hole。而这个hole一旦在开始的阶段没有被introduce，后来看Page Coloring的文章，就总是迷糊。
这个被教科书忽略或者miss掉的概念就是：Cache Bin。
Cache Bin是一个逻辑概念。不是CPU Cache机制里的任何实现；也不是OS内部数据结构里的任何Struct_。
Cache Bin是一个当OS或者以后要谈的应用程序利用大Cache的时候，做映射时，产生的一个Cache SET的集合。
每一个Cache Bin其实就是一个所谓的颜色（Color）。
下面我们试图对Cache Bin做一些略微学术化的定义。
Cache Bin： 在基于SET/Way的高端CPU中，给定（Given）一块连续的OS或者应用程序分配的物理内存，会按照一定的映射算法，落在（Fall Into）一块连续的Cache SET中。这个连续的Cache SET 集合，我们称之为：Cache Bin。
用类BNF范式，我们可以这样来定义：
[Cache Bin]::={Cache SET}^M(对于给定大小的物理Page，例如4K，其Cache Bin所Cover的Cache SET的M是固定的，例如，上节所case study的128个SET）
[Cache SET]::={Cache Way}^N (N means N-Way Cache）
在OS和CPU Cache之间引入一个Cache Bin的概念之后，对OS和CPU Cache的整体关系图，就与经典的OS和体系结构有了一点区别。这个区别是一个逻辑上的理解的区别，而非任何物理的实现：

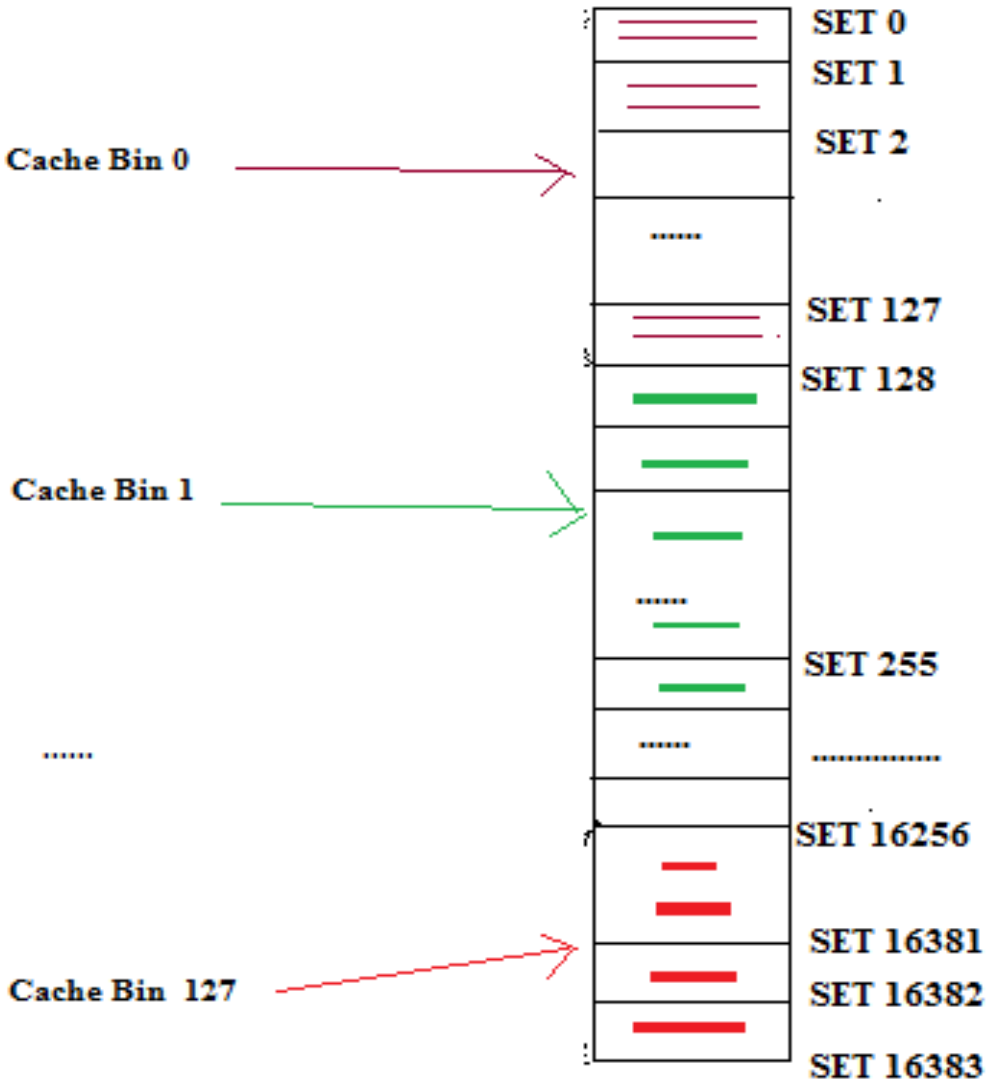


上述体系结构图基本上是把握高端CPU Page Coloring的总图。从物理内存上面，就是OS的Memory Management的Allocator或者应用程序，例如，通信系统的Data Path的内存管理了。然后通过一个个的mapping，最后落得某一个包厢（Set)的长凳子（Way）上。

这样OS与CPU之间的语义就存在了一定程度的Aware了。

下图所示为在上几节中，given 一个2M的4Way SET-Assoc L2 Cache，4K的物理

Page的case study中，Cache Bin的分布情况illustration：

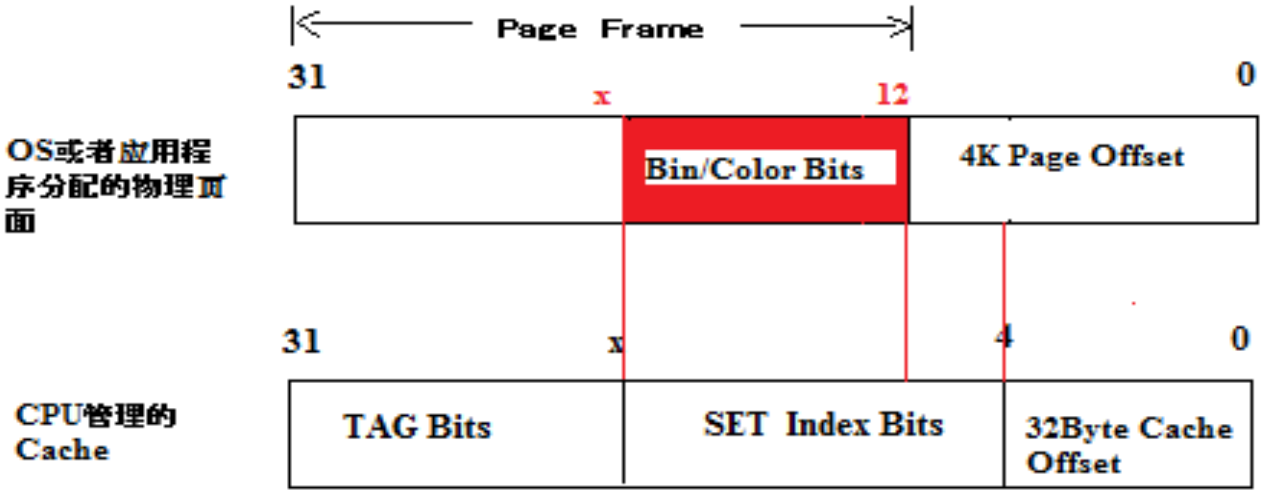


陈怀临注：每个Cache Bin就是一系列SET的集合。每个Cache Bin对应，或者说，就是，这一个颜色。当然我们引入Cache Bin这个“逻辑”概念后，一个物理内存，例如一个4K Page，首先是落入到“属于自己的”Bin里【找到妈咪给您分配的区域了】。然后再所属的包厢区域中，找到属于自己的SET（包厢）。人满为患？领班小姐（LRU算法）为您请出一个家伙俩，让您潇洒，600元RMB一个小时。。。

浅谈高端CPU的Cache Coloring（5）

上几节中，笔者试图介绍并强调了Cache Bin在理解大CPU大Cache中Cache Coloring的关键作用。读者最关键要理解的是： Cache Bin是当OS或者应用程序分配数据结构时，

最后映射Cache这个层面中出现的一个“逻辑”概念，而非CPU中存在一个类似的的电路
【Cache对 SET中TAG和内容的寻找，是具体的电路的，通过一个Local Bus寻址。】

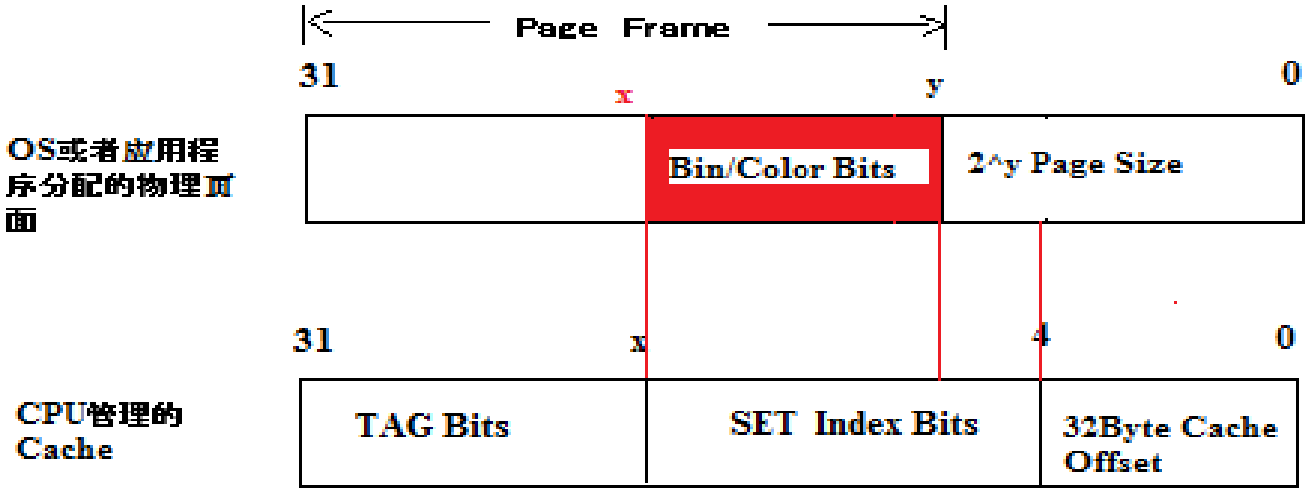


陈怀临注：(x－12＋1)就是所谓的Color Bits。或者说是Cache Bin的Index Bits。这几个bits的value决定了，given 一个4K大小的物理页面，在一个给定大小的Cache下，其去那个Cache区域(Cache Bin)。

上图所示的就是两个层面的映射图。
在Cache中，这个x的位置或者值取决与Cache的大小。
在上几节中，这个x的值是18.
为什么？
“
现在考虑一个L2 Cache，参数如下：
–大小：2M
–SET/Way：4 Way SET
–Line: 32 Bytes
这个2M的L2 Cache有多少SET（集合或者组）？
*2M/（32 × 4）= (2 × 2^20)/ (2^5 * 2^2)=2^14=16K=16*1024*
=10240+6144=16384
”
一个cache line是32byte的cache有2^14个SET,意味着：需要14个bit来贡献给CPU的local bus电路来做寻址。
大家知道，0-4bit是32bytes cache line的offset。所以，贡献的bit是从5开始的。

因此， x-5+1=14–》x=19-1=18.
换言之，一个32位的物理地址的5-18是SET的bits。
在上图的第一个部分是一个物理页面的分解图。0-11是4K的offset。 5-31是Page Frame.
现在把Cache的分解图平移上去，我们会发现一个（x-12+1）的红色区域。
这个红色区域就是：Color Bits。或者说，Cache Bin Bits。再或者说，决定了上节所描绘的一个4K的物理页面会去那个Cache Bin。
同学们，这个时候要非常注意了：Page Frame是OS或者应用程序在做物理内存分配时可以控制的。
换言之：OS和/或应用程序可以控制Color Bits，或者Cache Bin，从而把一个物理内存，有意思的（Intentionally) 布局。
这类似与，中央或者机关的部长可以指派手下的小部长或者经理们去外研所当个钦差大臣。喜欢的去美国；不喜欢的去古巴。或者反之。总之，权力就是春药。
也也类似于，妈咪看见一个大款来了，或者公安局长来了，一定是去西厢。好好伺候。如果是个煤老板；让他去东厢。应付一下算了。
总之，有了Cache Bin，就预备了权力，具备了预算，有了权力，就可以调配了。
下面我们来做一些case study：
当x=18的时候，18-12+1=7。
这就是本文中的例子的情况。系统有2^7=128 Cache Bin。
当x=19的时候，显然是256个Cache Bin。
x=19是什么意思？
如果仍然是一个4Way和32Byte的Cache，其意味着是一个4M的Cache。
依次类推，
当x=20的时候，是一个4Way/32byte的8M Cache。系统有9个color bit，有2^（20-12+1）=512个Cache Bin区域。
当x=21的时候，是一个4Way/32byte的16M Cache。系统有10个color bit，有2^（21-12+1）=1024个Cache Bin区域。
当x=22的时候，是一个4Way/32byte的32M Cache。系统有11个color bit，有2^（22-12+1）=2048个Cache Bin区域。
同学们这时会问一个问题了。。。
陈首席你现在，up to now，讲的都是经典的4K Page。这已经out of date了。。。
现代操作系统和CPU都已经支持大页面（Big Page）的allocation了。
那么Page Coloring或者Cache Coloring还适合嘛？
Yes，kids。
其实数学就是一个抽象。学术就是玩符号。

我们把12这个fixed的value变成y。
我们就得到了下面这张General的图：



Cache Bin/Cache Color Bits 模型图

陈怀临注：(x-y+1)就是Color Bit或者Cache Bin的Index，2^(x-y+1)就是相应Page条件下，系统的Cache Bin的个数。如果(x-y+1)<0, Cache Bin个数为0.

在这个通用模型下，
Number of Color/Bin Bits
b= (x-y+1).
Number of Bins
BIN = 2^B , if b>=0;
BIN = 0 , if b<0;

例如，如果一个操作系统或者应用程序分配的物理页面上512K。其y的值就是18.
下面来做一些定量讨论：
如果系统还是我们文章中例子的2M Cache。其x是18.
(18-18+1)=1.
这意味这什么？ 这意味着，一个2M的cache在OS或者应用程序的512K的页面分配机制下，cache其实就是被拆成了两个Bin。（2^1=2)
任何两个512K的物理页面或者数据结构，一不小心就在一个Bin里。
512K的数据结构在大型通信系统里比比皆是。
有多少人想过，两个512K的数据结构有可能是在互相残杀？？
例如，如果一个操作系统或者应用程序分配的物理页面上1M。其y的值就是20.

如果系统还是我们文章中例子的2M Cache。其x是18.
（x-y+1）=(18-20+1)=-1.
这意味这什么？ 这意味着，一个2M的cache在OS或者应用程序的1M的页面分配机制下，cache bin其实已经失效。就是一个bin了。
大家随便踩了。 OS或者应用程序脱离对Cache Friendly的任何控制。
换言之，如果想通过应用程序或者OS对Cache能够进行干涉，要确保：
x>=y.
从而，（x-y+1）>=1.
从而 2^(x-y+1)>=2.
从而确保系统中至少存在2个Cache Bin区域。

浅谈高端CPU的Cache Coloring（5）

前面5节阐述了大CPU大Cache的Cache Page-Coloring的一些基本概念。主要是通过妈咪的包厢运作制度来作为各位弟兄们比较熟悉的场景，从而达到融会贯通的。
在Cache Page-Coloring方面，一个前提是“里面放了N个长凳子的包厢制度”–Set Associative的Cache。否则，一切无从谈起。
这一节谈一下目前大CPU中的L3 Cache的一些问题。通常我们说LLC–Last Layer Cache。
目前市场上不少芯片都有了On-Die的L3 cache，例如Intel的Nehalem-EX, Westmere，IBM的Power7，RMI的XLP，Tilera等等。
与L1和L2的Cache相比，L3 cache的设计和管理方式有相同的地方，也有不一样的地方。
这里比较容易犯迷惑。
1。L3也是Set / Associative的。换言之，学术界和工业界没有，也没有傻到整一个新的Cache机制。从而，Page Coloring的各种思想和算法是而且当然是可以apply到L3 cache中去的。
2。 与L1，L2 cache controller相比，L3的实现是各个厂商差别很大。各有千秋。但总体而言，都是通过一个Distributed L3 Cache的机制，在系统层面提供一个完整的Set / Associative的L3 Cache。其中用Ring结构的还是比较多，例如Intel，RMI等等。Tilera用是其声称的Mesh结构。
3。对L3 cache理解里最容易出错误的就是这个通常不是Local bus的Interconnect结构，例如Ring。从而对其寻址方式和其结构方式造成混淆。请记住：Local Bus的直接拿Index bit来寻址和通过Pa做Hash来寻址，这是一个微结构的事情（Micro - Arch），而非结构

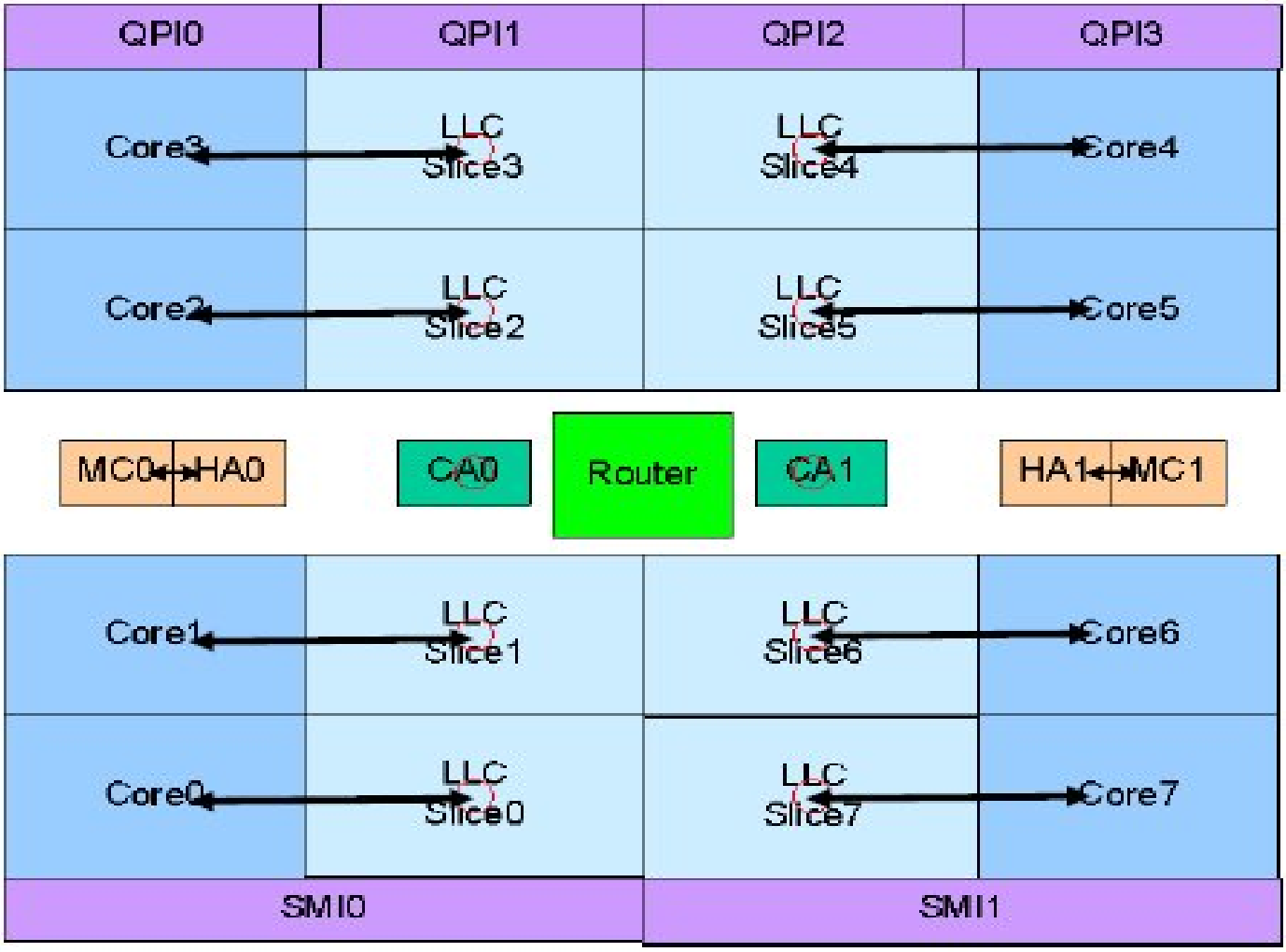
（Arch）的事情。在结构上，L3仍然是一个 Set / Associate的Cache。例如Nehalem - EX（Beckton）的24M的24Way的L3 Cache。这就说明，这个L3 Cache有(24 * 2^20)/(24 * 2^6) = 2^14=64K Sets.

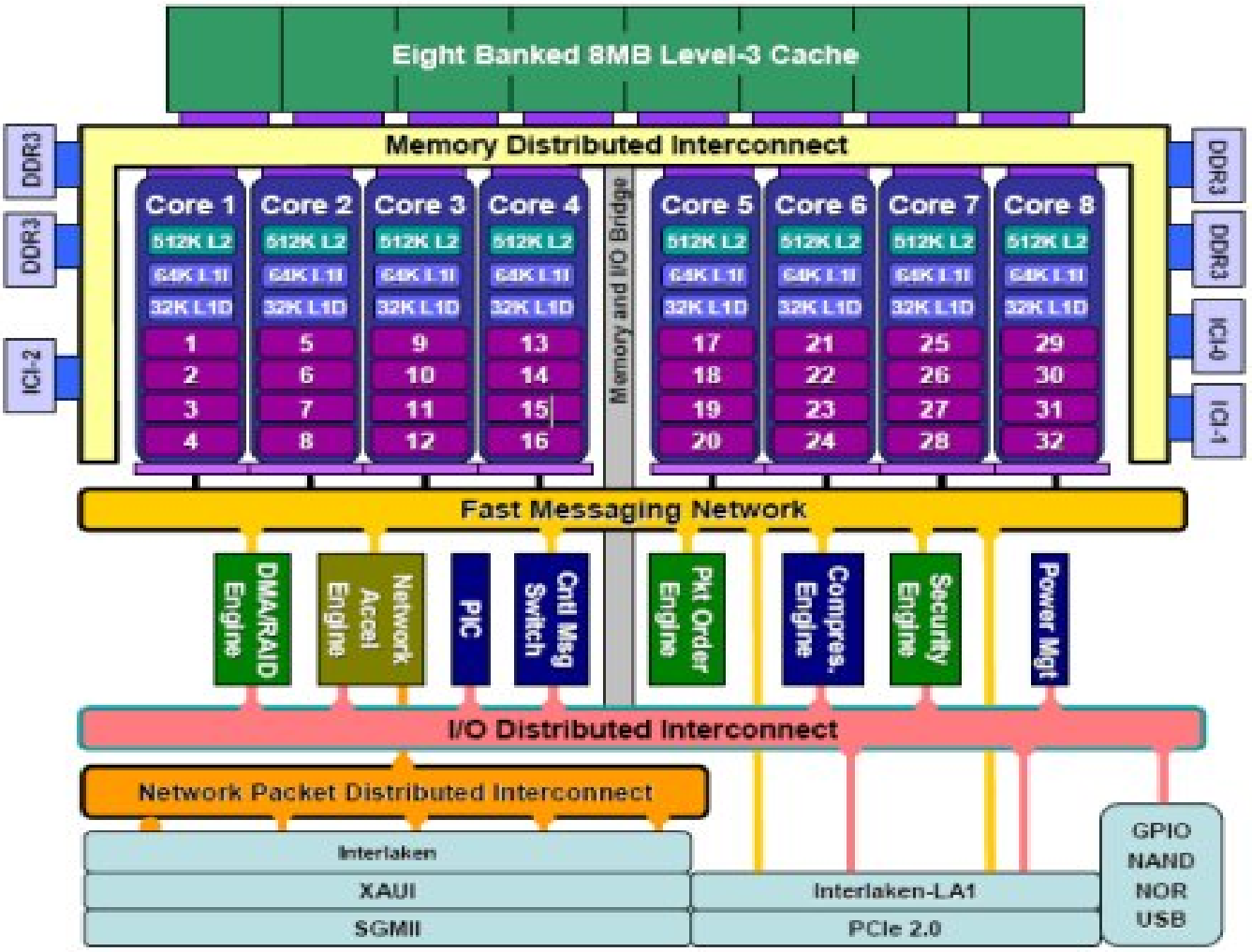
[Nehalem的Cache Line是64Bytes]。

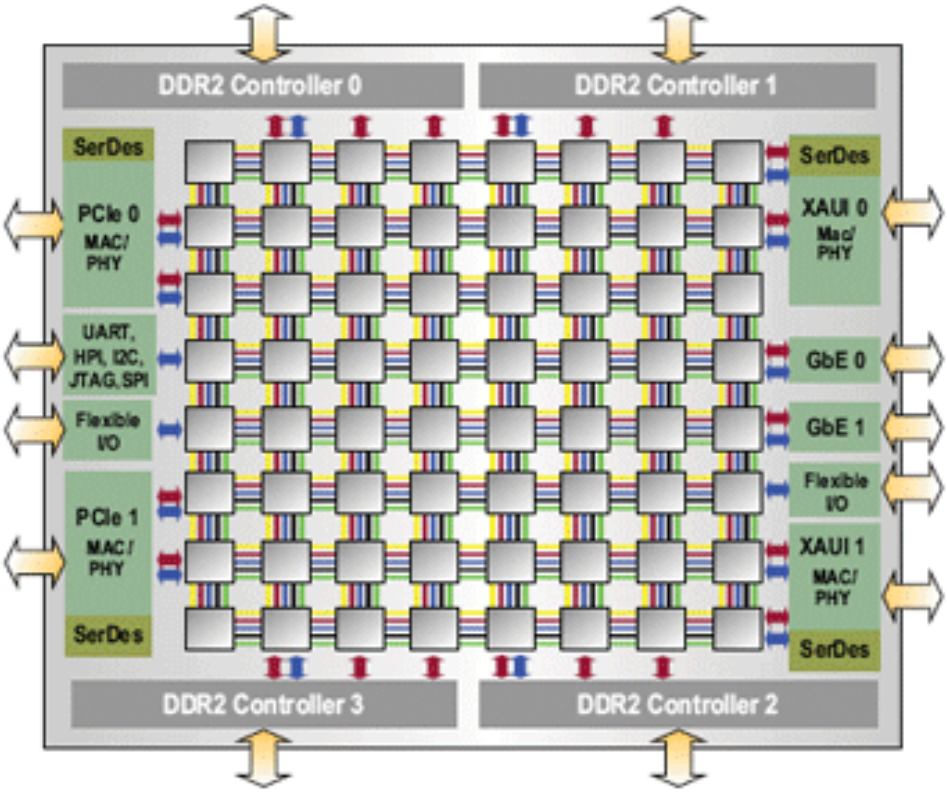
显然，通过我们上述几节的分析，在Intel CPU上，一个44位物理地址的0 - 5是cache offset；6 - 19是Set的Index bit。我们会在后续文章中对Intel的L3 Cache的微结构做一些探讨，例如其hash机制。

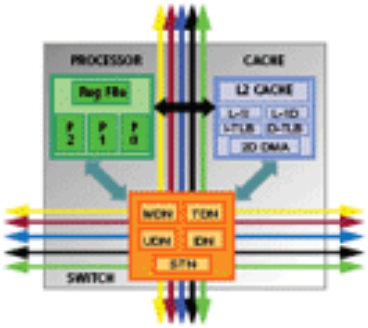
下面是一些相关CPU（Nehalem - EX；XLP - 832

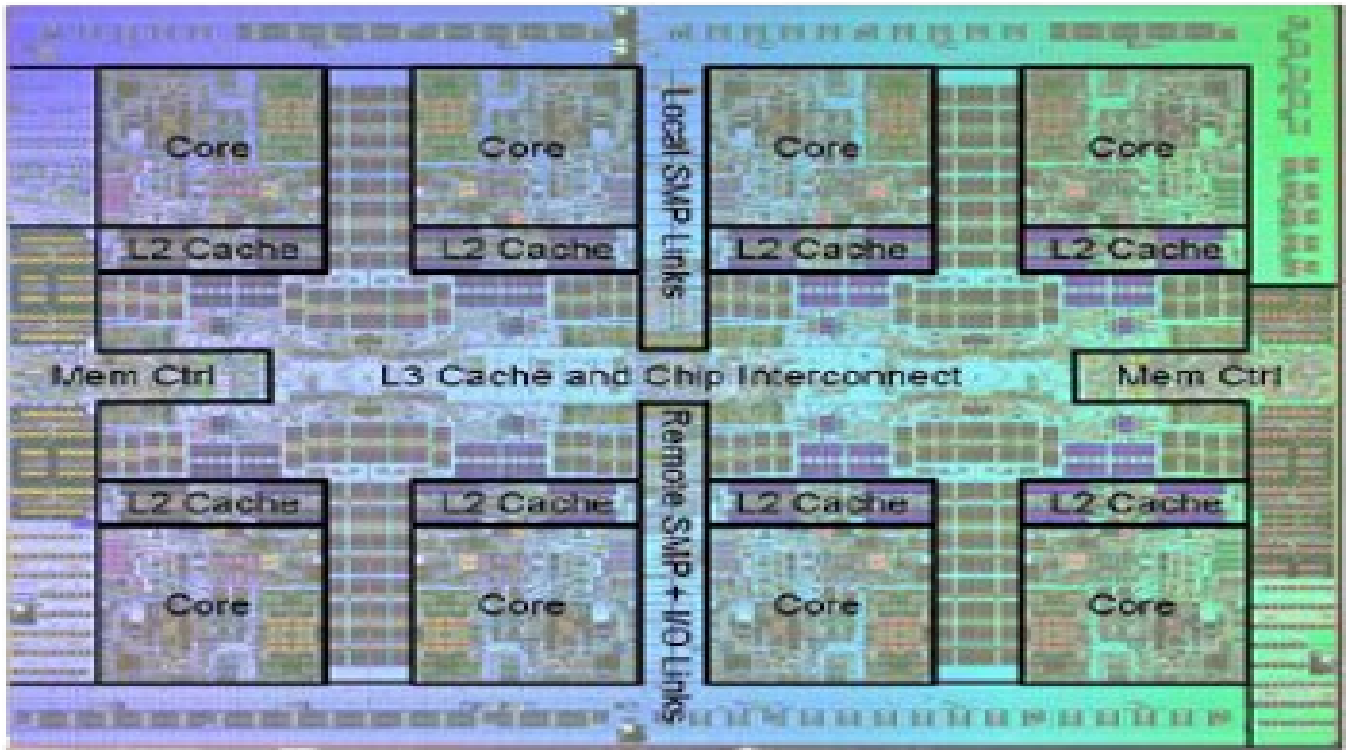
，Tilera和Power7）的L3 Cache结构示意图：




Eight Banked 8MB Level-3 Cache
Memory Distributed Interconnect
DDR3
DDR3
ICI-2
DDR3
DDR3
ICI-0
ICI-1
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
Core 8
512K L2
64K L1I
32K L1D
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
Memory and I/O Bridge
Fast Messaging Network
DMA/RAID Engine
Network Accel Engine
PIC
Cntl Msg Switch
Pkt Order Engine
Compres. Engine
Security Engine
Power Mgt
I/O Distributed Interconnect
Network Packet Distributed Interconnect
Interlaken
XAUI
SGMII
Interlaken-LA1
PCIe 2.0
GPIO
NAND
NOR
USB







L3 Cache的许多微结构的细节涉及到许多公司的商业机密。例如，Hash函数的实现分布等。本文就不没事找事，过分详细描述了。

In summary，L3的分布需要理解的一点是： LLC本身其实就是一个Bin的实现。相邻的Cache Line在L3里的分布是散落在各个LLC的Slice里的。每个Slice本身是一个SET / Associative的。

对L3最大的把握是：在支持ccNUMA系统中，例如QPI，一定要试图使得计算能力close to cache和memory。